Tutoriat SO 10: Main Memory

January 9, 2026

Contents

1 Background and Contiguous Memory Allocation

These are very well described here so go there to learn about it, it's a very important part of how memory works!

2 Fragmentation

This was discussed in Tutoriat 8 so go check it out.

3 Paging

Paging is a memory management scheme that allows a process' physical memory to be non-contiguous, which solves the issue of **external fragmentation** and eliminates the need

for **compaction** (the action of moving all currently running processes to one side of the memory, grouping all the tiny memory holes into one large, usable block).

The CPU works with **logical (virtual) addresses**; programs believe they have a single, contiguous block of memory starting from 0. In reality, a program is scattered across a number of **physical memory addresses**.

A program is divided into fixed-size blocks of logical memory addresses, also known as **pages**. The physical memory (RAM) is also divided into blocks (of the same size as pages), known as **frames**. These two types of blocks must be of the same size, since frames are used to store pages.

The **memory management unit (MMU)** is a physical hardware component in the CPU, whose job is to translate logical addresses into physical ones. A **page table** is a data structure stored in RAM that lists which page corresponds to which frame. Logical addresses are divided into two components:

- **Page number (p):** used as an index into a **page table**, which contains the base address of each frame in physical memory. The corresponding frame number is extracted from the page table.
- **Page offset (d):** the frame number and the offset together comprise the **physical address**.

The **page size** (like the frame size) is defined by the hardware and is typically a power of 2. If the size of the logical address space is 2^m and a page size is 2^n bytes, then the high-order $m-n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Typically, pages are 4KB or 8KB in size.

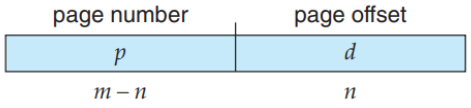

Figure 1: Periodic task

Since the OS manages physical memory, it must be aware of the allocation details: which frames are available, allocated, how many total frames there are, etc. This information is generally kept in a single system-wide data structure known as a **frame table**.

Each process maintains a copy of the page table, and a **page-table base register (PTBR)** points to it. Changing page tables requires only changing this register.

The **translation lookside buffer (TLB)** is a high-speed cache (between 32 and 1024 entries) that stores recent virtual-to-physical address translations. When the CPU needs to access a memory location, it first checks the TLB; if the page number is found, its frame number is immediately available, resulting in a **TLB hit**; otherwise, it's a **TLB miss**, and a new entry can be added. If the cache is full, an existing entry must be selected for replacement. However, some TLBs allow certain entries to be **wired down** (cannot be removed from the TLB).

Some TLBs store **address-space identifiers (ASIDs)**, which uniquely identify processes. This way, the TLB can contain entries for multiple processes simultaneously, so that every time a new page table is selected, the cache does not have to be erased.

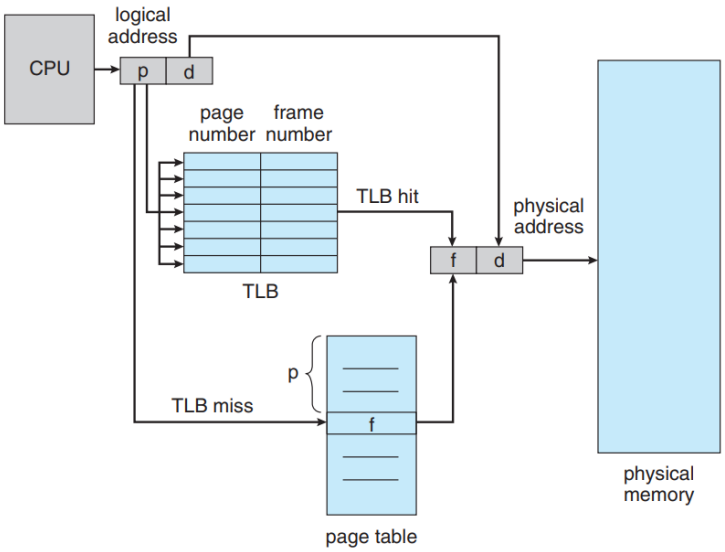

Figure 2: Paging hardware with TLB

3.1 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. One bit can define a page to be read-write or read-only. At the same time a physical address is being computed, its protection bits are checked to verify that no writes are being made to a read-only page; otherwise, a hardware trap happens.

One additional bit is generally attached to each entry in the page table: a **valid-invalid bit**. When the bit is set to valid, the associated page is in the process' logical address space and is thus a valid page. When the bit is set to invalid, the page is not in the process' logical address space.

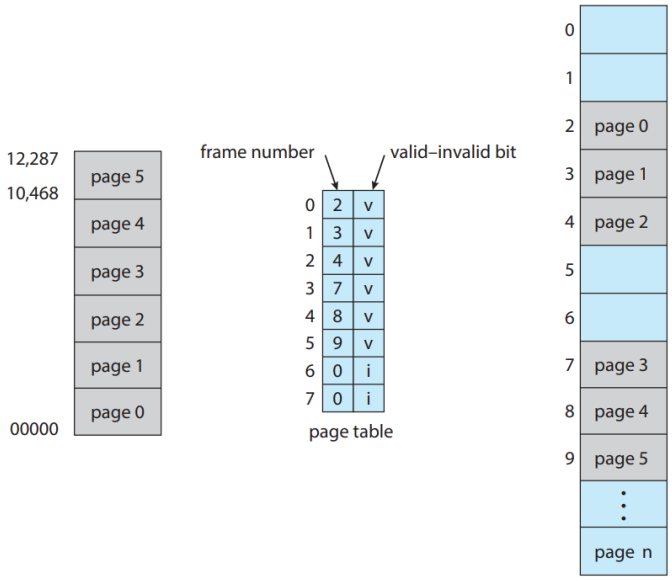

Figure 3: Valid (v) or invalid (i) bit in a page table

3.2 Shared paging

Processes can share pages for common code (their page tables contain certain pages for code that never changes during execution).

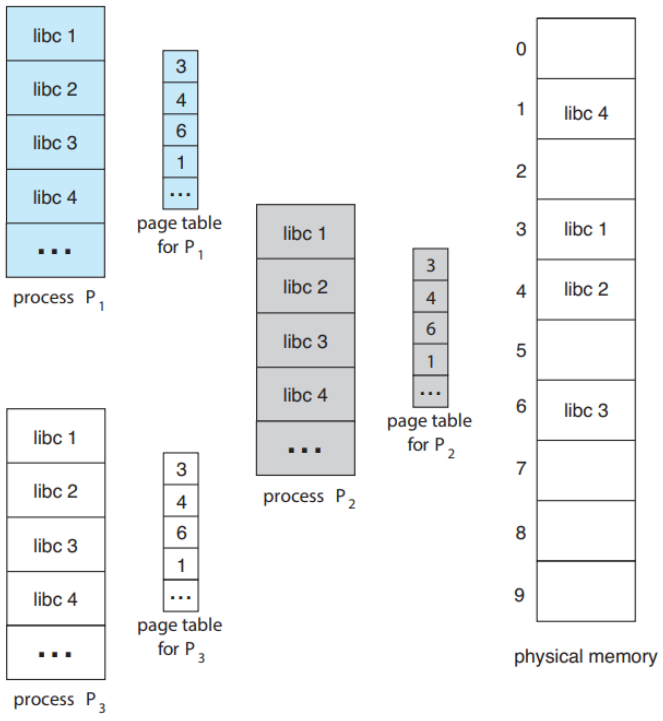

Figure 4: Sharing of a standard C library in a paging environment

4 Structure of the page table

4.1 Hierarchical paging

If the page table becomes too excessively large in memory, a solution would be to divide it into smaller pieces. One way would be to use a **two-level paging algorithm**. The page number is divided into two numbers: one for indexing into the "outer table", and one for the "inner table" (inner page number). This can be taken further (three-level paging, four-level, etc.).

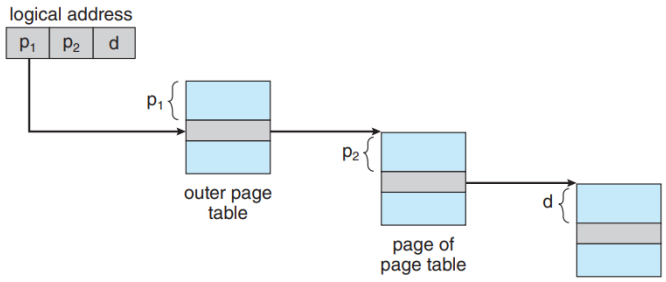

Figure 5: Address translation for a two-level 32-bit paging scheme

4.2 Hashed page tables

For address spaces larger than 32 bits, a **hashed page table** can be used, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements (chaining).

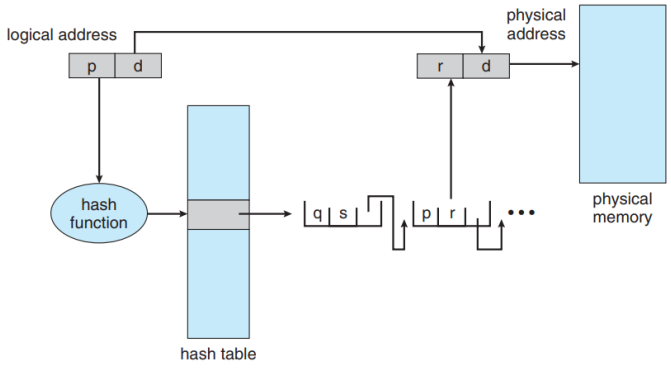

Figure 6: Hashed page table

A variation of this scheme has been proposed, which uses **clustered page tables**; they are similar to hashed page tables, except that each entry in the hash table refers to several pages (instead of a single page). They are typically useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

4.3 Inverted page tables

Usually, each process has an associated page table. The issue is that these tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

A solution could be using an **inverted page table**, which will be the only page table in the system, and will have one entry for each page of physical memory. In traditional paging, the question is "I have a process. Where in RAM are its pages located?"; in inverted paging, the question becomes "I have a frame. Which process and which page does it hold?".

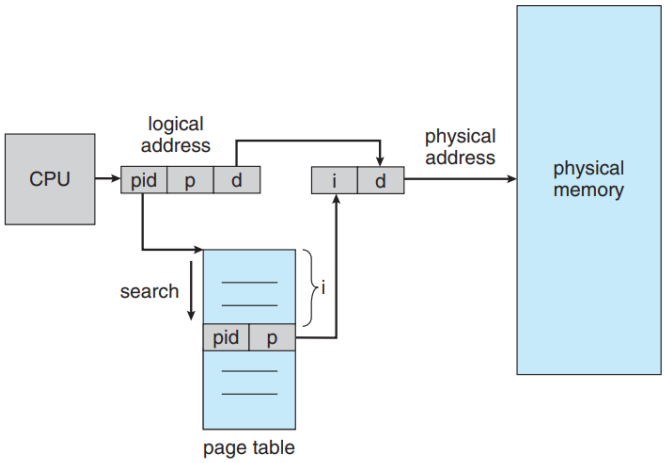

Figure 7: Inverted page table

5 Swaping

Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

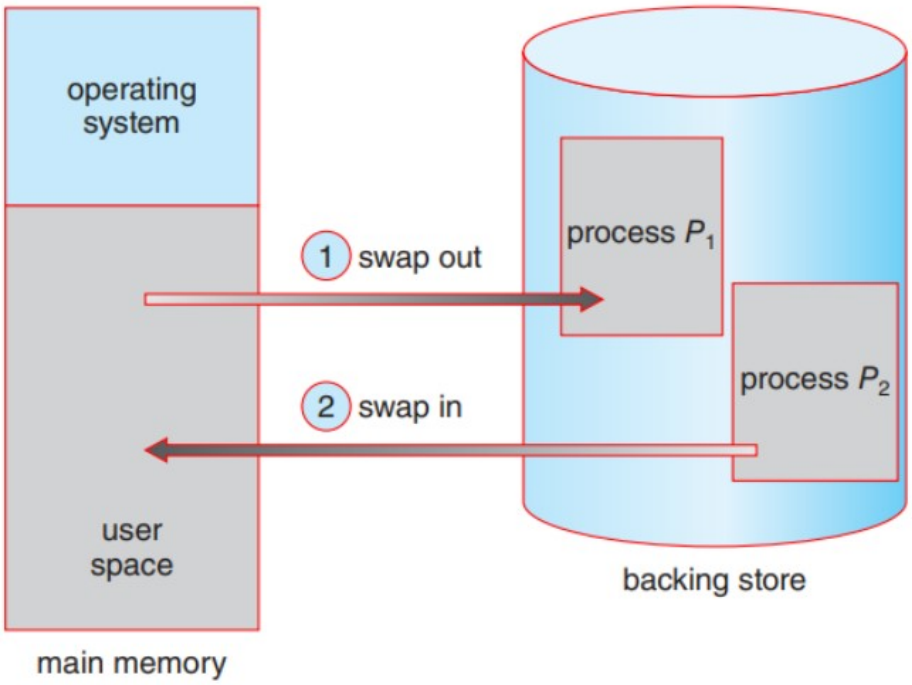

Figure 8: Standard Swapping Diagram

There are two main swapping approaches that we are going to discuss:

5.1 Standard swapping

Standard swapping involves moving entire processes between main memory and a backing store. The backing store is commonly fast secondary storage. It must be large enough to accommodate whatever parts of processes need to be stored and retrieved, and it must provide direct access to these memory images.

When a process or part is swapped to the backing store, the data structures associated with the process must be written to the backing store. The advantage of standard swapping is that it allows physical memory to be oversubscribed, so that the system can accommodate more processes than there is actual physical memory to store them.

Idle or mostly idle processes are good candidates for swapping; any memory that has been allocated to these inactive processes can then be dedicated to active processes. If an inactive process that has been swapped out becomes active once again, it must then be swapped back in.

5.2 Swapping with paging

Most systems, including Linux and Windows, now use a variation of swapping in which pages of a process—rather than an entire process—can be swapped. This strategy still

allows physical memory to be oversubscribed, but does not incur the cost of swapping entire processes, as presumably only a small number of pages will be involved in swapping.

In fact, the term swapping now generally refers to standard swapping, and paging refers to swapping with paging. A page out operation moves a page from memory to the backing store; the reverse process is known as a page in.

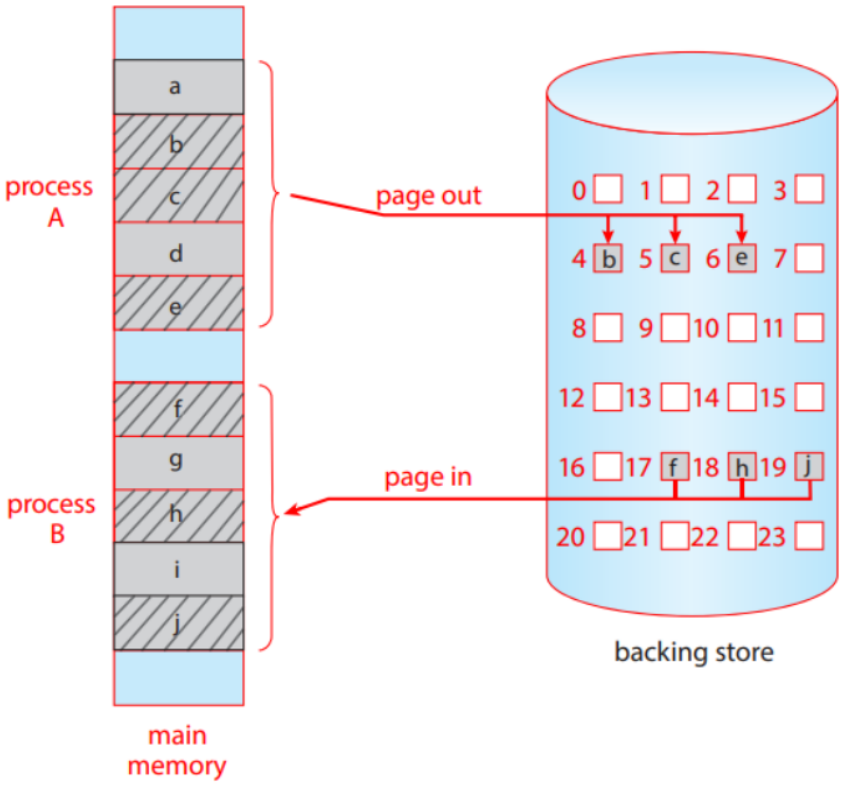

Figure 9: Paging Strategy Diagram